A Message-Passing Model for Highly Concurrent Computation

Alain J. Martin

Computer Science Department
California Institute of Technology

Caltech-CS-TR-88-13

A MESSAGE-PASSING MODEL FOR HIGHLY CONCURRENT COMPUTATION

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena CA 91125, USA

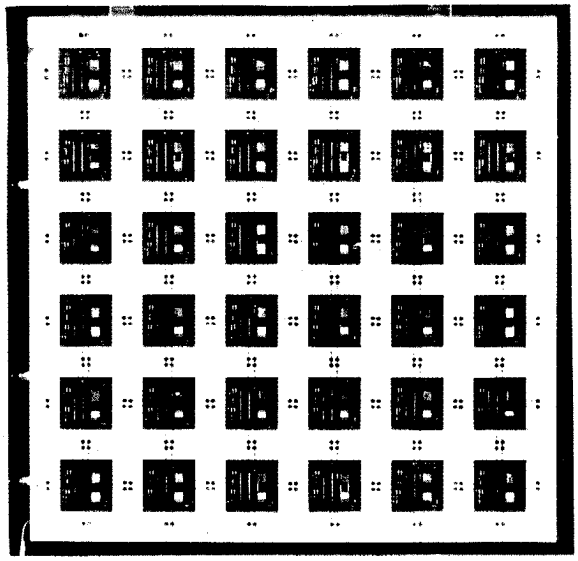
1. Introduction

The photograph of Figure 1 shows a prototype of a highly-concurrent distributed computer, built by the author at the Philips Research Laboratories in the Netherlands. Begun in early 1979, the design was completed in five months and debugged in a few days; it worked flawlessly until the UV-erasable memories began to show signs of amnesia. The total effort had taken less than two man-years. The hardware was a torus-like network of 36 "nodes," each connected to four neighbors. Each node was a simple but complete computer consisting of a processor chip with 1 kbyte read-only storage, and a 256-byte random-access-storage chip. The system was fully distributed: no common storage, common communication channel, or common clock. Communication and synchronization were achieved by message exchange among neighbor nodes using a bit-serial half-duplex protocol. Each node contained a simple operating system for the implementation of communication, process interleaving, and storage allocation.

The system was based on the programming paradigm of concurrent processes communicating by explicit exchange of messages. The most remarkable aspect of the model—at that time—was that the mapping of a distributed computation on the network did not require the programmer to know the size of the machine: the computation could grow and shrink dynamically through the network as if the network were infinite.

In spite—or because—of its simplicity, the "Torus" was a dramatic demonstration of the feasibility of the model as a general-purpose distributed computer [1],[2]. The crucial steps towards a full-blown machine were taken by C.L. Seitz at Caltech with the construction of a 4-node "Cosmic Cube" in 1982, and a 64-node version in 1983[3].

To appear in *Proc. 3rd Conf. on Hypercube Concurrent Computers and Applications*, 19-20 January 1988, California Institute of Technology - JPL, Pasadena CA 91125



–Figure 1–

Based on essentially the same computational model as the Torus, the cube uses a binary n-cube network instead of toroidal mesh, with the requested hardware upgrading of the nodes: a more powerful processor with a floating-point co-processor, a larger storage, full-duplex asynchronous channels. A version of C, extended with communication primitives is available as programming language, together with the required support software for loading the programs into the nodes.

An important difference with the Torus, however, is the availability in the Cosmic Cube of a deadlock-free protocol for routing messages between any two nodes. The Torus was built on the premiss that locality of communication should be maintained by the programs; such a system is called a *processing surface*. The Cosmic Cube drops the locality requirement by providing a routing protocol between arbitrary nodes.

The Cosmic Cube prototype proved so successful that the design was licensed and rapidly commercialized (1985). Several manufacturers are already offering "cube-machines" which are routinely used by researchers—at Caltech and elsewhere—to solve computationally demanding scientific problems [4].

We first formulate the problem of constructing distributed computations in terms of mapping a "computation graph" on an "implementation graph". We next introduce the program notation which we illustrate with some examples. We then discuss the mapping issue: process interleaving, communication, routing. Finally, we discuss the choice of a network.

2. Computation Graph and Implementation Graph

In [5], the problem of designing distributed computations has been formulated as a two-stage process. In the first stage, a computation is designed as a set of *communicating processes* connected by *channels*. A *process* is a sequential computation described by a sequential program operating on local variables, and by communication actions on channels. Processes do not share variables: they communicate with each other and synchronize their activities by means of communication commands. Such a computation is said to be *distributed*.

In more traditional forms of concurrent processing, the set of processes is statically defined. But in an environment where the potential concurrency is very high, we prefer to let the concurrency of the computation vary according to the size of the problem in much the same way as the depth of a recursion or the number of steps of an iteration. We therefore introduce the possibility to create and destroy processes dynamically. A concurrent computation is regarded as a graph, called the *computation graph*, in which the vertices are processes and the edges are channels. The computation graph may grow and shrink during execution according to the needs of the computation. At this point, the correctness of the computation should be guaranteed independently of the structure of the machine on which the computation is to be mapped.

In the second, mapping stage, the freedom to create an unbounded number of processes has to be reconciled with the finiteness and the topological restrictions of the distributed architecture. Such an architecture is also represented as a (now fixed and finite) *implementation graph*, whose vertices are the *nodes* of the machine and whose arcs, the *links*, are the physical communication channels between the nodes. In fact, the computation graph formalizes the needs of the computation in terms of active components and their topology. The implementation graph formalizes the constraints imposed upon the computation by a given architecture; that is, the possibly unbounded number of processes has to be mapped on a finite number of processing and storage units with limited and fixed communication channels among them.

3. The Program Notation

Several languages fit into the general programming model we propose. For the purpose of this paper, we describe the main features of a notation inspired by C.A.R. Hoare's CSP [6]. We therein describe only the communication primitives, which differ from those of CSP.

Communication

Communication has two semantic functions: It provides a distributed assignment statement and a synchronization mechanism. A communication consists in the synchronized executions of an input action in a process, say, p, and an output action in another process, say, q. The possibility for the two processes to communicate directly is represented by an edge from q to p in the communication graph. Such an edge is called a channel and is directed from the process performing the output to the process performing the input. A channel is shared by exactly two processes. As we shall see in the examples, two processes may share several channels.

There are several reasons for introducing channels. First, since we are describing a graph, we want to introduce both the vertices and the edges. Second, since we want to replicate identical processes, it is more convenient to describe a process in isolation with its dangling channels, and to compose several instances of the process by abutment of channels. Third, whatever notation is used, it is necessary to distinguish from each other the different "services" that processes require from other processes. In particular, it may be important not to order messages related to different services. Associating a channel with each type of service is a convenient solution to this problem. Fourth, the use of several channels provides a means to bypass the ordering of messages inside one input queue, since messages on different channels are not ordered.

Execution of input and output actions on the same channel have to be synchronized, since a message cannot be received before it has been sent. The *slack* of a channel is defined to be the difference between the number of completed output actions and the number of completed input actions on the channel at any point in time. Let $s(C)$ be the slack of channel C, the synchronization requirement is to maintain the relation $0 \leq s(C) \leq SB(C)$ invariant, for a given integer constant $SB(C)$ called the *slack-bound* of C.

There are three types of communication primitives according to the value of the slack-bound, namely: (1) $SB = 0$, or (2) SB positive and finite, or (3) SB infinite, (see [7]). In the first case, the channel has no buffering capacity: At any time, the number of messages sent equals the number of messages received. In the second case, the channel may buffer at most SB messages that have been sent and not yet received. In the last case, the channel may buffer an arbitrary number of messages. From a synchronization viewpoint, commands of the third type behave like P and V operations on semaphores. For what concerns synchronization, the three types of communication commands are semantically equivalent.

There is an important difference, however, for what concerns message passing: In the case of type (3) primitives, a channel must be able to buffer an unbounded number of messages; If possible, it would make it possible to implement a Turing machine!

First-in-first-out channels

We have already mentioned that the communication commands implement a distributed assignment statement: the nth message received on a channel is the nth message sent on the channel. To enforce this property in the case the channel has a non-zero buffer, it is necessary that the messages be received in the same order that they are sent, i.e., the channels have to be first-in-first-out.

Channel selection

The input and output commands defined above are sufficient for programming entirely deterministic processes. But we want to allow for some form of nondeterminism. For instance, we want to be able to construct a process p communicating with two other processes by the input channels X and Y. At some point, p is to receive a message either along X or along Y, nondeterministically. In that case, we need a mechanism to select the channel on which an input action is *firable*, i.e., the execution of the action does not violate the synchronization requirement .

Even in scientific computations, non-determinism may be introduced by variations in speed and differences in convergence rates or termination behavior.

In the Torus, we defined a special form of input action, called *selection*, operating on a given set of input channels—in the example above, the set would be $\{X, Y\}$—and on a so-called *channel variable*. Completion of the selection amounts to completion of an input action on one of the channels of the set on which a communication is firable. When several channels can be selected, the choice is nondeterministic. The identity of the selected channel is recorded in the channel variable.

Since any selection mechanism requires to test whether an action is firable on a given channel, we can as well provide a boolean command that does just that. Such a command, called the *probe*, has first been proposed in [8], and is used in the Cosmic Cube. With communication commands of the first type, as in the original definition of the probe, the probe on channel X, denoted here by $\&X$, means "an action is pending on X". With commands of the third type, as in the Cosmic Cube, the probe on X is defined only if X is an input channel, and means "the buffer of X is not empty".

In our experience, the combination of the probe and input and output commands on channels provides a simple and yet general set of communication primitives. Let us postpone introducing the mechanism for the dynamic creation of processes, and present some examples first.

4. Examples

Since we have not yet discussed the mechanisms for creating processes, we assume in the examples that a computation graph of the size and topology required by the input parameters of the problem has already been created. We just describe the internal structure of a process. (The first two examples present very little interest from the point of view of parallel algorithm design. They will be used solely to illustrate the trade-offs between different communication primitives. The solution of third example, however, is believed to be new.)

Example 1: Nearest-neighbor computation

An array x occupies the points of a two-dimensional grid: variable $x_{i,j}$ at point (i,j). The initial value of x is xin. The final value of x is obtained by nearest-neighbor iteration: For an internal point (i,j), the next value of $x_{i,j}$ is computed as the function f of the current values of $x_{i,j}$, and of its four neighbors $x_{i-1,j}$, $x_{i,j-1}$, $x_{i+1,j}$, and $x_{i,j+1}$. The iteration is supposed to converge to a fixed point. This example is characteristic of many numerical applications, e.g., the computation of Laplace's equations by finite differences.

A process is attached to each grid point (i,j) to compute the value of $x_{i,j}$ at that point. We describe a process p corresponding to an internal point of the grid. From now on, x represents the element of x at that point, and xin its initial value. We use local names for channels: p is connected to its north neighbor by output channel No and input channel Ni , to its south neighbor by output channel So and input channel Si. No and Ni in p are identical to Si and So in the north neighbor, respectively. The connections with the east and west neighbors are similar.

Let us say that we are not sure about the convergence of the algorithm, and that we want to look at the result after 10000 iterations. The main body of the program—we omit declarations and simplify initialization—is as follows.

```
1: i:= 0; x:= xin;
2: *[ i<10000 --> No!x; So!x; Eo!x; Wo!x;
3:                Si?s; Ni?n; Wi?w; Ei?e;
4:                x:= f(x,n,s,e,w);
5:                i:= i+1
6:  ]
```

Observe that since all processes are identical, a slack-bound equal to zero would lead to deadlock: All processes start sending their current value of x but none can receive it. It is easy to see that a slack-bound of 1 is necessary and sufficient to avoid deadlock, and that the slack of any channel is at most 2, independently of the slack-bound selected. We can also arrange the processes in the manner of a black-and-white checkerboard: the black processes use the above program; the white processes use a program derived from the above one by interchanging lines 2 and 3. In that case, deadlock is avoided even with slack-bound zero for all channels.

Example 2: The knapsack problem

Given are a non-empty list l of natural numbers, sorted in non-decreasing order, and a natural number s. We want to determine the truth of the predicate: "There exists a set of elements of l whose sum is equal to s". The predicate is denoted by $K(s,l)$ in the sequel. (The problem is known to be NP-complete.)

We shall use the functions $n(l)$ giving the number of elements in l, $hd(l)$ giving the first element of l, and

$tl(l)$ giving the list obtained by deleting the first element from l. The last two functions are defined only if l is not empty. The solution is based on the well-known *divide-and-conquer* method, using the following property $(x = hd(l))$:

$$K(s,l) = \begin{cases} true, \text{ if } s = x; \\ false, \text{ if } s < x \ \vee \ s > x \wedge n(l) = 1; \\ K(s-x, tl(l)) \vee K(s, tl(l)), \text{ otherwise.} \end{cases}$$

The problem is ideally suited for a recursive solution: Either $K(s,l)$ can easily be evaluated directly or it requires the evaluation of $K(s-x,tl(l))$ and $K(s,tl(l))$. The main effort in constructing a distributed algorithm goes to implementing a distributed procedure call. The processes are connected in a binary tree. In the following program, each process receives from its "father" process the parameters s and l [line 1] and sends to its father the boolean result b [line 9]. If two new values of K have to be computed, the process sends a pair of parameters to its left "son" and a pair to its right "son" [line 6], and receives a boolean answer from each of them [line 7].

```
1: Fi?(s,l);
2: x:= hd(l);
3: [ s=x                          --> b:= true
4: | s<x or s>x and n(l)=1 --> b:= false
5: | s>x and n(l)>1        -->
                  Lo!((s-x),tl(l)); Ro!(s,tl(l));
6:                Li?b1; Ri?b2;
7:                b:= (b1 or b2)
8: ];
9: Fo!b .
```

A slack-bound zero is sufficient to avoid deadlock. The length of the list sent by a process to its sons is one less than the length of the list the process received. Hence, if n is the length of the initial list, we need a complete binary tree of processes of depth n, i.e., containing $2^n - 1$ processes, to execute the algorithm in n steps. Given a binary tree of processes of depth k, $0 \leq k \leq n$, the complexity of the algorithm is of the order of $k + 2^{n-k}$.

Example 3: Small bag of integers

We want to implement a bag—or multiset—of at most $2^n - 1$ integers as a complete binary tree of depth n, each process containing one element of the bag. The root of the tree communicates with the user of the bag by means of four channels—*PUT*, *GET*, *HAS*, and *RES*—using the following four operations:

- $PUT?x$: x is added to the bag. (A *PUT* action is never attempted when the bag is full.)
- $GET!x$: The smallest element of the bag is returned in x and removed from the bag. If the bag is empty, then the special value NIL is returned.
- $HAS?x; RES!b$: b returns the number of occurrences of x in the bag.

Each process communicates with its "right son" along the channels $PUTR$, $GETR$, $HASR$, and $RESR$, and similarly with its "left son". Each process that is not a leaf of the tree behaves as the user of the bag implemented in each of its subtrees. For reasons of efficiency—we want to achieve a complexity of the order of n for each of these operations—we require that:

1) for each subtree, the smallest element of the bag implemented in the subtree be kept in the root of the subtree;
2) the tree be balanced. For each subtree t, if $left(t)$ and $right(t)$ denote the number of elements in the left and the right subtrees of t, respectively, we maintain:

$$0 \leq left(t) - right(t) \leq 1.$$

In order to do so, we introduce a boolean g in each process p wich is not a leaf, and maintain the invariant:

$$g \equiv (left(t) - right(t) = 1)$$

where t is the subtree rooted at p.

Initially, all g's are *false*, and all x's are NIL, where NIL is a constant chosen larger than any element in the bag. The programs for a leaf process and for a non-leaf process follow. Again, a slack-bound zero is enough to avoid deadlock.

```
Leaf process:
1: *[[ &PUT --> PUT?x
2:   | &GET --> GET!x; x:= NIL
3:   | &HAS --> HAS?y;
                [x=y --> RES!1 | x<>y --> RES!0]
   ]] .

Non-leaf process:
4: *[[ &HAS --> HAS?y;
5:              [ y<x --> RES!0
6:              | y>x --> HASL!y; HASR!y;
                          RESL?a; RESR?b;
7:                        [y=x --> RES!(a+b+1)
                          |y>x --> RES!(a+b)
                          ]
                ]
8:   | &GET --> GET!x; GETR?r; GETL?l;
                x,y:= min(r,l),max(r,l);
9:              [ y=NIL  --> g:= false
10:             | y<>NIL --> [ g --> PUTR!y
                              | not g --> PUTL!y
                              ];
11:                          g:= not g
12:             ];
13:  | &PUT and x=NIL  --> PUT?x
14:  | &PUT and x<>NIL -->
                        PUT?y;
15:                     x,y:= min(x,y),max(x,y);
16:                     [ g --> PUTR!y
                        | not g --> PUTL!y
                        ];
17:                     g:= not g
18: ]] .
```

Communication behavior

The three examples, which we believe to be typical, show that the choice of communication primitives is not critical for the construction of the programs. Once the first program has been transformed in the "checkerboard" fashion, all three programs work equally well independently of the value of the slack. Observe that, even when an infinite slack is used, the number of messages in a channel is at most two: the slack is bounded by the tight synchronization among processes.

We also observe that communication actions tend to occur in sequences: eight in a row in the first an example, and up to five in a row in the last example. Furthermore, the order among actions of a sequence is not specified by the problem. Worse, choosing an arbitrary order may lead to deadlock, whereas executing the actions of a sequence concurrently would not. This is the case in the first example, where executing all 8 actions concurrently does not require slack 1 or a checkerboard organization of the processes.

Systolic and diffusing computations

In the first example, once the processes are started, their activity is a continuous alternation of of internal computations and communications with the four neighbors. A computation with such a behavior has been called a "systolic computation". In the second example, the activity diffuses from the root of the tree towards the leaves, and then shrinks from the leaves towards the root. We call a computation with such a behavior a "diffusing computation". But, if the first example were complete, the computation would also contain the passing of the parameters—the initial values of x—and the collecting of the results—the final values of x. In that case, it would also contain a diffusing phase: The parameters are propagated through a spanning tree of the computation graph—called the diffusion tree—starting at a process (the root of the diffusion tree) which is connected to the input device and diffusing towards the leaves of the tree. It would also contain a shrinking phase: the results have to be collected following the edges of a spanning tree in the direction of its root, i.e. a process that is connected to the output device. (In general, the two spanning trees are identical.) Hence any complete computation comprises a diffusing phase, a systolic phase, and a shrinking phase. (A computation sometimes consists of a repetition of these three phases.)

5. Process Creation

In most computations, different input parameters lead to different computation graphs. (In the Knapsack example, the maximal depth of the tree is equal to the number of elements in the set, and the precise size of the tree depends on the elements in the set.) Hence, it may be difficult or even impossible to determine the size of a computation graph before execution. It may also be inefficient to do so, since not all processes of a computation graph are always active at the same time. We shall therefore include the possibility to create processes during the computation.

Creation of a process p may be postponed until that point in the computation where another process fp reaches an output action on a channel linking p and fp. We call this technique "lazy creation". Another description of the technique is that the processes are created but left "dormant" until the first input action activates them. A dormant process uses no storage. With this technique, the creation of processes follows the diffusion tree. Processes obey the commandment *"Thou shalt procreate, but thou shalt not kill"* according to which processes may create other processes in a hierarchical way, but do not kill other processes—instead, a process "dies" after completing the last action of its program.

Clearly, creation of a process implies its placement in a node. Once process p has been placed in node n, the information about the pairing (n,p) has to propagate to all direct neighbors of p, since these processes will need the information in order to communicate with p.

Three different mapping and placement strategies can be considered:

- *Static creation and static placement* The computation graph is fixed and placement is decided before execution of the program, either by the programmer or compiler. All processes are created and placed during initialization of the computation. A simple initialization process in each node reads in the programs, and creates and starts the processes. All initialization processes are linked in a fixed tree that is a spanning tree of the implementation graph rooted at one of the nodes that are connected to the input devices.
- *Dynamic creation and static placement* Placement is determined prior to execution by the programmer or compiler. Processes are created dynamically when they are first sent a message by another process. Since the placement of each process is decided before execution, the placement information of each process is available to all neighbors of the created process. The drawbacks of this technique are that, 1) dynamic load balancing is excluded, and 2) all computation graphs have to be subgraphs of the same "closure graph"—e.g., all binary computation trees are subtrees of the infinite complete binary tree. However, we think that the advantage of this approach—simplicity—outweighs its drawbacks.
- *Dynamic creation and dynamic placement* Creation of processes is "lazy", and placement is decided during the computation. Placement strategies can deal with load balancing and arbitrarily modifiable computation graphs. The overhead involved in the placement algorithm and in the routing of placement information has to be weighed against the generality of the method.

6. Implementation Issues

The concern for keeping the processors (nodes) usefully busy may rapidly become self-defeating if it leads to solutions in which a large portion of the processor time is spent in such overhead activities as load balancing, process

migration and scheduling. We have adopted the strategy: "the simplest scheduling is no scheduling", supported by what we call a "saturation policy": The process grain is small enough and the problem sizes are large enough that the number of processes for any non-trivial problem is at least an order of magnitude larger than the number of nodes. Consequently, any reasonable placement strategy guarantees that each node is saturated, i.e., it contains enough processes to be active most of the time.

Hence, the implementation issues are:

- The interleaving of processes in one node.
- The implementation of communication.
- The routing of messages between arbitrary nodes.

Interleaving of processes in a node

Let us assume that we have chosen to implement the second mapping strategy (dynamic creation and static placement). After compilation, each process has acquired a global name, the global names of its neighbors, and their addresses (node numbers). For an arbitrary node n, we shall describe how the a priori concurrent activities of the processes inside n are simulated by interleaving. We still make no assumption about the implementation graph.

Since the activity of a process is an alternation of computation and communication, we assume that a node contains one or several *computation processors* and one or several *communication processors*. For the sake of simplicity, we assume here that there is just one processor of each type. The generalization is straightforward. The computation processor executes sequences of computation actions. The communication processor executes sequences of communication actions.

The set of processes inside n is partitioned in two disjoint sets: the set of computing processes—processes currently executing a computation sequence , and the set of communicating processes—processes currently executing a commmunication sequence.

The identities of the computing processes are kept in a priority queue, called the "computation queue", according to some priority scheme, so as to guarantee that each process is eventually selected for execution. At any time, the first process in the queue is the currently running process, i.e., the process currently executed by the computation processor. The identities of the communicating processes are kept in a similar queue—the communication queue.

The execution of the running process proceeds until the process either terminates or reaches a communication action. In both cases, the next process in the queue, if any, becomes the running process. In the case where a communication action has been reached, the running process is removed from the computation queue and added to the communication queue. Upon completion of the communication, the process is put back in the computation queue. The interleaving of the communication processes is exactly symmetrical.

The interleaving strategy just described introduces restrictions on the computation concurrency: Concurrent processes that are mapped in the same node can no longer proceed concurrently, since their activities are interleaved. The interleaving restricts the class of possible states reached by the implemented computation. But the class of states reached is still a subset of the class of states reached by the implementation-free computation. Hence the implementation does not introduce deadlock if (i) the creation of a new process is always possible, i.e, there is enough storage space inside the nodes; and (ii) the implementation of communication does not introduce deadlock.

Implementation of communication

Let us consider the implementation of a matching pair of communication actions $C!(exp)$ and $C?(var)$ in processes X and Y, respectively. C is the global name of the common channel; exp is the expression of X the value of which is to be assigned to variable var of Y. We assume the channel to have slack-bound zero.

According to the semantics of communication primitives with slack-bound zero, the communication can be completed if and only if both communication actions are pending, i.e. both X and Y have reached the action of the matching pair. In the implementation of $C!(exp)$ in X, the predicate "a communication action on C is pending in Y" is implied by the truth of the Boolean variable $q(X.C)$. The Boolean variable $q(Y.C)$ plays an equivalent role in the implementation of $C?(var)$ in Y. A possible implementation is as follows:

```
C!(exp) in X:

  1: q(Y.C) <-- true;
  2: [q(X.C)];
  3: q(X.C):= false;
  4: SEND(exp)

C?(var) in Y:

  5: [q(Y.C)];
  6: q(Y.C):= false;
  7: q(X.C) <-- true;
  8: RECEIVE(var)

&C in Y: q(Y.C)
```

The above implementation illustrates several important points.

- The first three commands of $C!(exp)$ and the first three commands of $C?(var)$ implement the synchronization part of the communication. Commands 1 and 7 are remote assignments called *signalling*. A signal from X on C [line 1] sets $q(Y.C)$ to true in Y. (Similarly for the command of line 7.) A signal is an unsynchronized communication action: A signal in X terminates independently of the state of Y. The command of line 2 is a *wait* action: Operationally, it stands for "wait until $q(X.C)$ holds". (Similarly for the command of line 5.)

- SEND and RECEIVE implement the distributed assignment of *exp* to *var*. When no message is transmitted, i.e. the communication is used for synchronization only, SEND and RECEIVE are omitted.
- From a correctness point of view, a running process need be interrupted only when it reaches a wait-action that cannot be completed and when it reaches a RECEIVE action, since the process that performs the corresponding SEND may be in the same node. From the point of view of efficiency, since the signal, SEND, and RECEIVE actions are likely to be time-consuming, a running process should delegate the execution of the whole communication action to the communication process.
- If type (3) communication primitives are used, the implementation of $C!(exp)$ and $C?(var)$ reduces to $SEND(exp)$ and $RECEIVE(var)$.
- When a computation process reaches a communication action, the placement information is used to determine the node address of the destination process and the link along which the routing should be initiated. If the destination process is in the same node as the source process, the communication is carried out by the processor.

As already mentioned, communication actions tend to appear in clusters inside a program. The same holds true for probes: In general, a set of guards containing several probes has to be evaluated at once. The overhead for useless process switching between the queues should be avoided between the actions of a cluster.

We need a mechanism to indicate the beginning and end of a cluster of communication actions. We can either let the compiler add the information or we can introduce communication primitives that operate on a set of channels. Such a set of primitives was used in the Torus. A process could send a message to several neighbors in any order—we called this action *broadcast*, and receive a message from several neighbors in any order—we called this action *collect*. A process could also select a true probe out of a set by an action called *select*.

But we prefer to be slightly more general and introduce a parallel construct || to indicate that an arbitrary set of communication actions has to be executed concurrently. For instance, we can use the notation $||(x1, x2, x3)$ to indicate that the three communication actions $x1$, $x2$, and $x3$ should be executed concurrently. The first example could be modified as

```
1: i:= 0; x:= xin;
2: *[ i<10000 --> ||(No!x, So!x, Eo!x, Wo!x,
                     Ni?n, Si?s, Ei?e, Wi?w);
4:                 x:= f(x,n,s,e,w);
5:                 i:= i+1
6: ]
```

In the implementation of the parallel construct, all component actions of the construct are initiated concurrently. The execution of the construct terminates when all components actions have been completed.

Avoiding incoming-message queues

The management of incoming messages by the node is one of the most serious problems encountered when infinite-slack primitives (unblocking sends) are used, since the number of messages to be buffered is unbounded. On the other hand, when finite-slack primitives are used, the maximal number of messages to be buffered for each process is known: It is the value of the slack, which has been declared by the programmer. It is therefore possible to allocate the necessary space in the state space of the process; intermediate buffering is not necessary. The advantages of this scheme, which was used in the Torus, should not be underestimated. It is probably worth the extra communication overhead. (We have seen that zero-slack communication primitives require three elementary communications actions *vs* one for infinite-slack primitives.)

It might be possible to have the best of both worlds if the programmer accepts the extra effort. The programmer uses infinite-slack primitives (non-blocking sends), but limits the effective slack of each channel—which occurred automatically in the three examples given. And he declares the effective slack in the programs. (The analysis necessary to determine the slack is already a part of the analysis for proving the absence of deadlock.)

Now, the corresponding buffer space can be allocated in the local space of the processes, as in the case of slack-zero primitives. And the incoming information is processed immediately: there is no need for the processes to rummage through messages queues to find the relevant information.

7. Communication Between Nodes

The parameters of a message are:

```
<destination_node,
 destination_process,
 channel_name>
```

in the case of a signal,

```
<destination_node,
 destination_process,
 output_value>
```

in the case of a SEND.

Upon receiving a message from a link, a communication processor, say, in node n, determines the destination node of the message. If the destination is n, the message can be immediately consumed by the destination process, i.e., it is stored in the already reserved space in the working space of the process. If the destination is not n, the *transit message* has to be forwarded to another node. The transmission of transit messages poses three problems:

- First, for each implementation graph, there must exist a simple routing function to determine the route of a message to any destination node.

- Second, the overhead in the transmission of a message, in particular the interference between the activity of the different link-processes involved in the transmission and the rest of the computation, must be carefully assessed and controled. This is probably one of the most difficult engineering issues in the design of such systems.
- Third, transit messages may completely congest the routing space of a node. It may happen that a cycle of congested nodes is formed in the implementation graph, and each transiting message in a node of the cycle has to be transmitted to the next node in the cycle. Such a situation is obviously a deadlock.

In the Torus, the form of deadlock just described cannot occur since there are no transiting messages, due to the locality requirement. In the Cosmic Cube, deadlock is prevented by using a routing protocol that avoids the formation of cycles in binary n-cubes.

A very elegant solution to the problem of transit messages has been proposed in [9], and implemented in the AMETEK machine described in [10]. It consists in separating completely the transmission of transit messages from the activity of the node by inserting a *message routing automaton* as an interface between each node and the network. With this solution, transit messages never enter the node.

8. Choice of the Implementation Graph

Thanks to the top-down approach we have followed, the choice of the implementation graph can now be guided by several criteria related to the different issues identified so far. These criteria are:

- The regularity of the graph and its extensibility.
- The possibility of mapping a large class of standard computation graphs—e.g., meshes, tori, trees, shuffle exchange networks—easily and efficiently.
- The existence of a deadlock-free routing protocol.
- A small diameter of the graph, so as to keep the routing distance between any two processes small.

It is very difficult to quantify the above criteria and to weigh them against each-other in a single figure of merit; however, it is clear that the binary n-cube meets all criteria very satisfactorily. Yet, the exponential nature of the binary n-cube, which is one of its advantages regarding diameter, mapping, and routing, may become a serious problem when the size of the network grows beyond, say, 64k nodes. At that point, the decrease in wire density necessary to pack the network in a three-dimensional space, will cause the communication latency of the network to increase significantly, compared to two or three dimensional networks like the torus or the mesh.

It has shown that it is possible to design an efficient deadlock-free routing protocol for the torus as well [9]. Again, the protocol requires introduction of a partial order on the links, and using a route that follows a decreasing path, so as to guarantee that no cycle can be formed. A priori, the torus does not contain enough links for such a protocol to exist; but since the half-links are associated to an output queue, it is possible to create virtual links by introducing extra output queues, and to impose a partial order on the virtual links.

This protocol makes the torus an efficient routing network. This property, combined with the density of large dimension tori compared to the density of large binary n-cubes, makes the torus a serious competitor of the cube when very large networks are considered, in spite of the better mapping performance and smaller logical diameter of the cube.

Acknowledgment

The research described in this paper was sponsored by the Defense Advanced Research Projects Agency, ARPA Order number 3771, and was monitored by the Office of Naval Research under contract number N00014-792-C-0597.

9. References

[1] Alain J. Martin, "The Torus: An Exercise in Constructing a Processing Surface," *Proc. 2nd Caltech Conf. on VLSI*, 527–537, Jan. 1981

[2] Alain J. Martin, "Distributed Computations on Arrays of Processors," *Philips Technical Review* 40, 8/9, 270–277, 1982

[3] Charles L. Seitz, "The Cosmic Cube," *CACM,* 28(1): 22-33, January 1985.

[4] Geoffrey C. Fox et al. , "Solving Problems on Concurrent Processors," Prentice-Hall, 1988.

[5] Alain J. Martin, "A Distributed Implementation Method for Parallel Programming," in Information Processing 80, S.H. Lavington (ed.), 309–314, North-Holland, 1980

[6] C.A.R. Hoare, "Communicating Sequential Processes." *Comm. ACM* 21,8: 666-677 August 1978

[7] Alain J. Martin, "An Axiomatic Definition of Synchronization Primitives," *Acta Informatica* 16, 219–235, 1981

[8] Alain J. Martin, "The Probe: an Addition to Communication Primitives," *Information Processing letters* 20: 125-130 1985

[9] William J. Dally, Charles L. Seitz, "The Torus Routing Chip," *Distributed Computing* 1(4): 187-196, Springer International,1986.

[10] Charles L. Seitz et al.. "The Architecture and Programming of the Ametek Series 2010 Multicomputer," *These proceedings*